

RIOT
A SIMPLE GRAPHICAL CHIP ASSEMBLY TOOL

Stephen Trimberger & Jim Rowson
California Institute of Technology
Pasadena, California 91125

TM #5004

TECHNICAL MEMORANDUM

**Department of
Computer Science**



RIOT
A SIMPLE GRAPHICAL CHIP ASSEMBLY TOOL

Stephen Trimberger & Jim Rowson
California Institute of Technology
Pasadena, California 91125

TM #5004

RIOT -- A Simple Graphical Chip Assembly Tool

KEYWORDS: Computer aided design; integrated circuits; interactive graphics; symbolic layout; stick diagrams; stretchable cells; routing; chip assembly; composition tools.

ABSTRACT: Errors in the chip assembly process are harder to find than errors in cell design, since they belong to no specific part of the design, but rather to the assembly as a whole. Assembly errors are more costly than cell design errors also, since they often go unnoticed until late in the design cycle. Interactive graphic tools typically require that assembly be done with primitive graphical operations, which are inappropriate for the assembly task. Language-based tools give more powerful assembly operations, but remove the two dimensional view of the chip necessary to visualize many assembly operations.

Riot is a simple interactive graphical tool designed to facilitate the assembly of cells into integrated systems. Riot supplies the user with primitive operations of connection -- abutment, routing and stretching -- in an interactive graphic environment. Thus, the designer retains full control of the design, including the assignment of positions to instances of cells and the choice of connection mechanism. The computer takes care of the tedious and exacting implementation detail, guaranteeing that connections are actually made. The powerful connection primitives give the user of Riot the ability to quickly assemble a custom chip from a collection of low-level cells. This document provides a discussion of the motivation for Riot and a description of the Riot chip assembly system, its capabilities and its use.

REFERENCE: TRIMBERGER, S and ROWSON, J. (1982) A Simple Graphical Chip Assembly Tool.

1 INTRODUCTION

Custom integrated circuit layout can be split into two major parts: *cell design*, the development of the low level cells making up the "leaves" of the hierarchical tree; and *composition*, assembly of those cells into larger cells and systems [Rowson 1980].

Current graphics systems address low-level cell design [Calma 1979], and rely on the simple, low-level graphical primitives needed in cell design to perform composition. Graphical systems allow blocks to be oriented and positioned, but composition requires operations to position one instance adjacent to another and logically connect connectors on instances. These features are usually unavailable in an interactive graphic system.

Powerful composition tools currently being investigated are language based. The composition of instances is specified by textual invocation and connection. Although the programming language form is very powerful, the textual specification has been found to be tedious and error prone in general systems, since composition continues to be primarily a graphical positioning task. Highly automated language-based systems allow designs to be expressed simply in a very abstract manner, but place severe restrictions on the floorplan of the resulting chip.

Riot is a simple interactive graphical composition tool. Riot provides primitives necessary for composition, so that designs can be made quickly and easily. Riot's primitives are powerful enough to be useful for composition, yet simple enough that control of the layout is still in the hands of the designer. Riot can read cells defined in Caltech Intermediate Form (CIF) [Sproull 1980], the Sticks Standard (Sticks) [Trimberger 1980], and its own *composition format*. The user can give commands to create an instance of a cell, and to connect cells by abutment, simple routing, or stretching. The output is in composition format which is translated to CIF for mask generation or Sticks for simulation.

1.1 Environment

Riot runs on the Caltech graphic workstation which consists of a "Charles" color terminal, a high resolution color raster display device; a CRT terminal; a Xerox mouse pointing device; and a Hewlett-Packard 7221A four-color pen plotter; all driven by a DEC LSI-11. Riot also runs on the low-cost GIGI workstation, which consists of a DEC GIGI color terminal and a Summagraphics BitPadTM pointing device (see figure 1).

Riot is written in SIMULA and runs on the Caltech Computer Science Department DECsystem-20. The SIMULA environment includes a powerful symbolic high-level language debugger and memory management facilities with dynamic memory allocation and a garbage collector. Riot consists of approximately nine thousand lines of code, including the shared low-level objects package (500 lines) and graphics package (4000 lines).

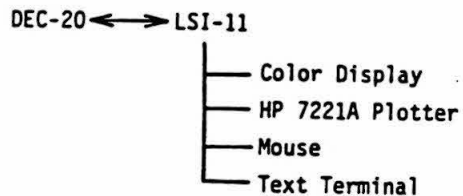


Figure 1a. "Charles" Terminal Color Graphic Workstation



Figure 1b. GIGI Terminal Color Graphic Workstation

The interface between Riot and all other design tools at Caltech is through three standard data formats: CIF, Sticks Standard, and Composition Format. CIF, a geometrical format, is generated by many design tools, including Bristle Blocks [Johannsen 1979], LAP [Lang 1979] and PLA generators. A user extension was added to CIF to indicate connector locations so that Riot's logical connection operations could be performed on CIF cells. CIF is also used for mask generation. Sticks Standard, a symbolic layout format, is generated by the graphical and textual symbolic layout systems, and is also used as input to simulation. The composition format is used by Riot to save an editing session. It contains a description of composition cells including the hierarchy description, locations of instances, locations of connectors on the composition cells, and references to files which contain the leaf cells used in those compositions.

1.2 Riot Definitions

Riot deals with a modification of the restricted hierarchy called the *separated hierarchy*. [Rowson 1980]. The hierarchy is built of *cells*, which are collections of integrated circuit data. There are two kinds of cells in Riot: *leaf cells* on the leaves of the hierarchical tree, consisting of primitive geometry and instances of cells; and *composition cells* in the interior of the tree, which consist only of instances of other cells. Riot only manipulates composition cells, all others are considered leaf cells, which Riot cannot modify. Riot allows the user to assemble composition cells and leaf cells into larger composition cells.

A leaf cell in Riot is described by its minimum bounding box and its connectors. A connector consists of a location on or inside the bounding box of the cell, a layer, and the width of the wire that makes that connection. Composition cells consist of the bounding box, the connector information, and the instances of cells which make it up. An *instance* represents the contents of a cell placed at a given location with a specified orientation and array replication count. An instance of a cell is shown with its bounding box outlined and its connectors shown as crosses in figure 2. The size and color of the connector cross indicates width and layer of the wire making the connection inside the cell.

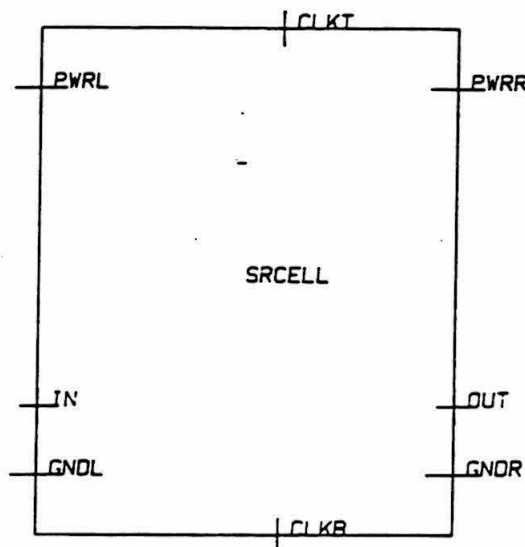


Figure 2. Riot's View of a Cell Instance with Bounding Box and Connectors

A *connection* is made by Riot by the placement of instances so that the connectors on the instances touch. Riot handles connection in the positional sense, not in the logical sense: a connection is the result of appropriate positioning. Therefore, once a connection is made, it can be easily (perhaps accidentally) destroyed. Riot does nothing to guarantee that connections will be maintained. This simplified handling of connection has limited the usefulness of Riot, as discussed later in this paper.

Riot's connection operations connect one instance to many instances, moving the one to touch the many so that the connections are made. A many-to-many connection can be made by defining a cell which contains one of the sets of cells, and connecting that one to the other many. Controlled overlapping of cells can be done in Riot, allowing adjacent cells to share a common power line, for example.

A *routed connection* or *route* is the sandwich made by interposing an instance of a routing cell, automatically generated by Riot, between two instances so that the routing instance connectors connect to one instance on one side and the other instance on the other side.

In many situations, a stretched connection uses less space than a routed connection. A *stretched connection* is made by modifying the symbolic layout Stick form of a cell so that a new cell is made with connectors located to connect to another cell without requiring routing. An instance of the new *stretched* cell is placed next to the existing instance to make the connection. Both routed and stretched connections are provided in Riot so the user can select the kind of connection that is more appropriate at each place in the design.

Riot manipulates only composition cells, so it deals only with cells and instances of cells. In Riot, the user may create instances of cells, connect those instances to one another, and define new cells which are made up of those instances and connections. Riot cannot make primitive mask geometry, except as a side-effect of a routing operation. Other tools exist which create the cells which Riot manipulates.

In Riot, a composition cell consists of a bounding box, a list of connectors, which includes connector name, layer, and position, and a list of the instances inside it. Instances are described by an instance name, the defining cell name, the replication counts and replication spacing of the array (if any), and the six elements of the transformation matrix which specifies the position and orientation of the instance in its containing cell.

Riot has two command interfaces: one textual, one graphical. The textual command interface, accessed with the keyboard, is used primarily to modify the editing environment. Textual commands get and put files, set plotting parameters, generate hardcopy plots of cells, set defaults for routing operations, and invoke the graphical command editor to modify a composition cell.

The graphical command interface is used to edit a cell and is accessed by pointing at items on the graphic display. The Riot display screen is divided into three pieces (figure 3): a large editing area next to two small menu areas along the right edge of the screen. The editing area shows the contents of the cell under edit. The upper menu area contains the names of the cells which are currently defined and which may be instantiated. The lower menu contains graphical editing commands which are invoked by pointing at them.

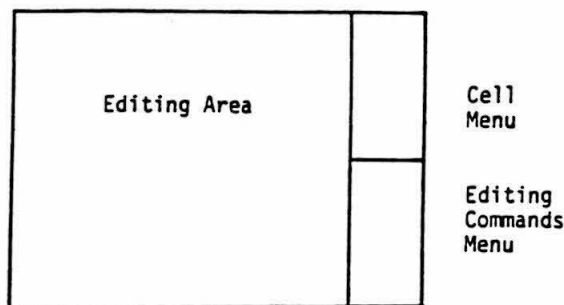


Figure 3. Riot Display Organization

2 RIOT OPERATIONS

Riot has commands for four different tasks: interface to the environment, creation of instances, connection of instances, and completion of a cell. This section discusses the four kinds of commands.

2.1 Interface to the Environment

Riot's interface commands access external cell definitions, generate hardcopy plots of cells in Riot, and modify the display and routing characteristics. Riot can read leaf cells defined in CIF or Sticks, and composition cells defined in composition format. Riot writes composition format files which can be converted to CIF for mask generation or to Stick for simulation. Since Riot is an interactive graphical tool, commands exist for zooming and panning the display. Commands also exist to set routing wire widths, to delete and rename cells, and to produce hardcopy plots of a cell.

2.2 Creation of Instances

A use of Riot creates an instance by selecting the cell in the cell menu by pointing at it, then giving the create command in the editing window. The user can optionally specify replication counts in x and y directions to create arrays, replication spacing for arrays, rotation by multiples of 90 degrees, and mirroring of the instance. When an array instance is created, interior array elements are abutted, but the replication spacing can be modified if desired by the user.

An instance is represented on the screen by its bounding box and connectors. The connectors need not be on the boundary of the instance. Optionally, instances can be displayed with their cell names and connector names inside the bounding box to facilitate identification. An array instance is treated like any other instance, and shows the gridding due to the replication of the cell in the array and all the connectors on the outside edges of the array. No access is provided to interior connections on arrays.

2.3 Connection of Instances

The main goal of Riot's commands is to facilitate making connections between instances, and this is where Riot differs from typical interactive graphic systems. Riot has simple positioning and orientation commands similar to those found in traditional graphics systems. These commands are used in Riot to prepare instances for connection but are not used to make the actual connection of instances. Riot has three connection commands which ensure that the connections will be made properly. Riot's connection commands connect instances by abutment, routing or stretching, guaranteeing connection and allowing the designer to select the connection method which best fits the design at that point.

Connection of instances is done in two parts: first, the connections to be made are specified, then the *connection specification command* is given to actually make the connection. A connection is a link from a connector on one instance to a connector on another instance. Riot checks that the connectors to be joined are on the same layer and that they are opposed. That is, that they connect top to bottom or left to right. For a given connection specification command, connections must originate at one instance but may connect to many instances. This eliminates possible ambiguities in moving many instances to optimal locations. If necessary, the *from* instance will be moved to match the *to* instance.

When a connection is specified by a user, the instances and the connectors of both sides of the connection are saved by Riot until the connection specification command is given. Then the connections are made and the logical connection information is thrown out.

Riot provides three connection specification commands, one for each of the kinds of connections which it can make: the user may specify merely that the instances are to be abutted, which is used if a cell has no connectors; the user may identify specific connections to be made; and the user may specify a bus-type connection in which all connections are made from one instance to another.

Riot also provides three fundamental ways to connect instances: abutment, stretching and routing. *Abutment* is used primarily if there are no connectors to guide the connection. Abutment makes the bottom or left edge match, depending on the relative positions of the instances before the ABUT command. If specific connections exist, Riot will do the abutment, attempting to match the specified connections (figure 4). If the connections cannot be made by the abutment, a warning message is produced. An option of the abutment command allows instances to be overlapped to share a common pair of connectors. This feature is frequently used to share power or ground lines in adjacent instances.

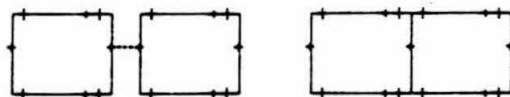


Figure 4. Connection by Abutment

In a *stretched* connection, the locations of the connectors on the *to* instance are used to determine the needed separations of the connectors on the *from* instance to make the connection by abutment. If the *from* instance is defined in symbolic form, the new constraints on the connector positions are put into the Stick file, making a new cell. The new cell is passed through the Stick optimizer in REST [Mosteller 1981], which moves the connectors to the constrained locations. Riot then removes the old instance and inserts an instance of the new cell into the cell under edit, abutted to the desired instance (figure 5). If the cell to be stretched is not a Stick cell, the stretch cannot be done and the command is aborted.

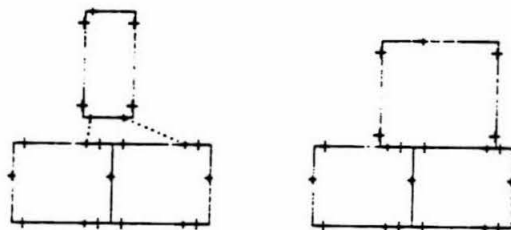


Figure 5. Connection by Stretching

Riot is able to make simple multi-layer *river-routed* connections. A multi-layer river-route is a routed connection between parallel sets of points where no routes change layers and no two routes on the same layer cross. The Riot river router cannot turn corners, and it ignores objects in the path of the route. These limitations are not too severe in the interactive environment provided by Riot.

When the route command is given, the connections on the *from* and *to* instances are used to specify starting and ending locations of the route. A simple algorithm makes a nearly optimal route between the sets of connectors specified. The wire widths of the route are determined by the widths of the connectors on the instances being connected. Riot makes a new cell containing the river route and places an instance of that route next to the *to* instances. The *from* instance is moved to connect to the other side of the river route instance, thereby using the least amount of space possible for the route (figure 6). If there is more than one instance routed to, and if the connectors do not form a straight line, Riot generates additional routes to fill the gaps and make all the connections. Optionally, the user may specify that a route be made without moving the *from* instance. This feature is used to make connections between two instances which are already positioned and should not move.

The routing cells made in Riot are treated just like other cells. They are entered in the list of cells, and may be instantiated, moved, and deleted by the user.

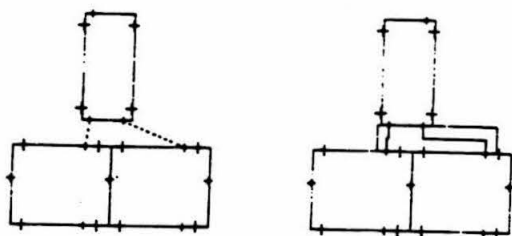


Figure 6. Connection by Routing

2.4 Finishing a Cell

A composition cell created by Riot includes those connectors from its instances which lie on its bounding box. The route command can be used to "bring out" connectors from the inside of the cell to the edge of the composition cell. When an attempt is made to route the connectors on an instance past the bounding box of the cell, a simple straight-line route cell is made for those connectors to the edge of the cell, and an instance of that cell is placed to make the connection.

3 RIOT EXAMPLE

The best way to understand the operation of Riot is through an example. This section takes a step-by-step walk through a Riot editing session, showing some of the features of the tool.

The chip being assembled is a four-bit sequential logical filter: a function defined on a series of inputs, x as

$$f_n = \sum_{i=1}^4 c_i x_{n-i}$$

where the c_i constants are supplied from off-chip and all sums and products are Boolean. A rough initial floorplan is shown in figure 7, showing how the designer wishes to lay out the design. This floorplan determines which cells are needed, how they must connect to one another and gives an initial guess at critical paths in the design.

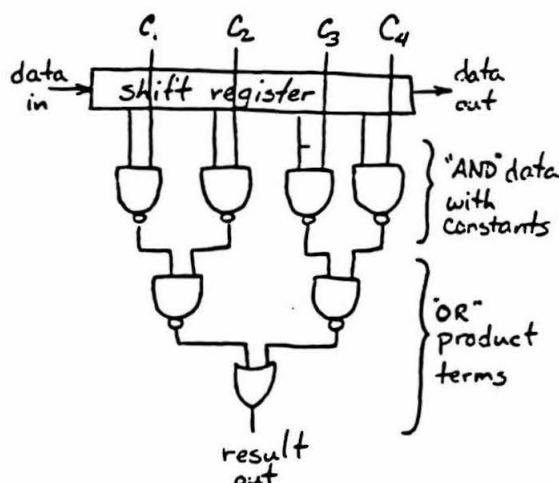


Figure 7. Rough Floorplan for Logical Filter

The cells to be assembled in Riot are shown in figure 8. The input and output pads were taken from a library of CIF cells, and are defined as geometry in CIF. The shift register cell, NAND and OR gates were laid out in REST, and are defined as symbolic layout in the Sticks Standard. Therefore, the pads cannot be stretched by Riot and all connections to them will have to be made by routing, but connections to the other cells can be made by stretching if the designer wishes to do so.

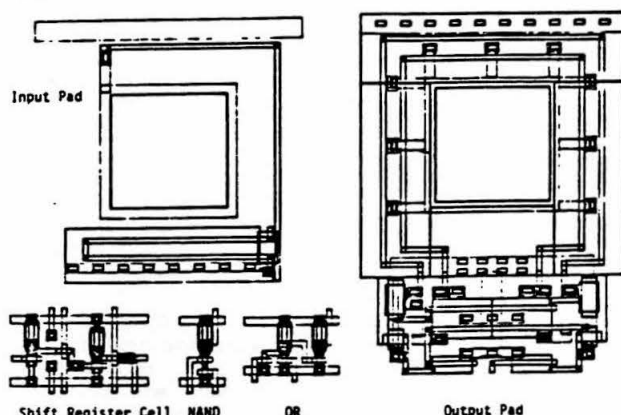


Figure 8. Leaf Cells for Logical Filter

The first step is to generate the shift register array. The array elements abut, making the shift register chain connections as well as power and ground connections. Next, two stages of NAND gates provide the ANDing of the constant terms and the first level of ORs, then routing is done to the OR gate. Connections to these gates are routed in figure 9a. Alternatively, the designer may save area by stretching the NAND gates, eliminating the routing area then adding the last OR stage (figure 9b).

In figure 9, the shaded areas are routing areas. The relative dearth of routing area in the stretched version is misleading, since some of the area is wasted inside the cells. The important space savings in the vertical direction since no routing channels are needed to connect the NAND and OR gates.

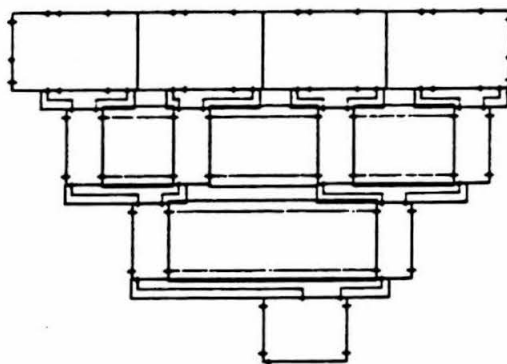


Figure 9a. Logical Filter Logic Connected with Routing.

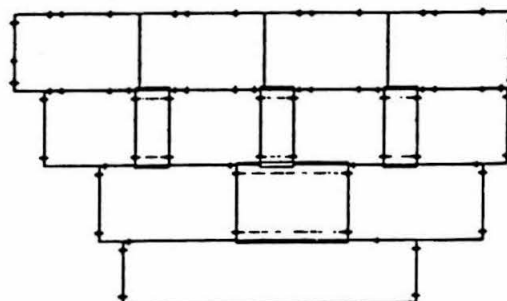


Figure 9b. Logical Filter Logic Connected with Stretching.

The definition of the logic portion is finished by routing connections to the edge of the cell so they show as connectors on the larger cell. Pre-defined pipe fittings aid complex routes for power, ground and clock lines to an instance of the logic cell. Pad routing is done in pieces with Riot's routing command, and the completed chip geometry is shown in figure 10.

This logical filter chip is, of course, of insignificant complexity, and is used only to show Riot's capabilities. However, the same techniques that allow this small custom chip to be assembled quickly can be used to significantly shorten the assembly time for larger custom chips.

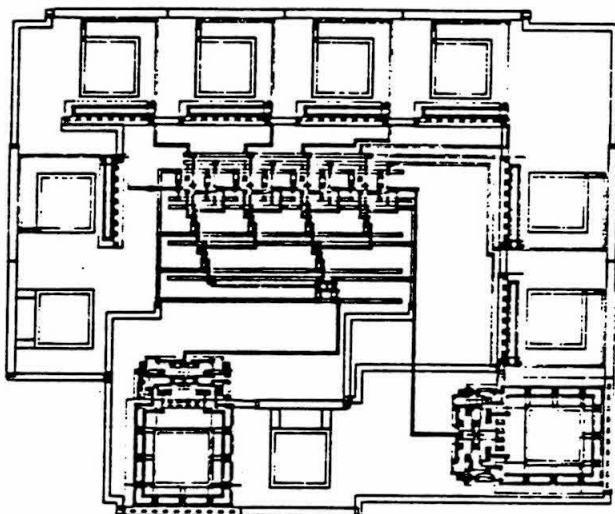


Figure 10. Logical Filter Chip Geometry

4 OBSERVATIONS, LIMITATIONS AND CONCLUSIONS

Riot was developed very quickly with shortcuts taken to simplify the software design. Riot represents about four man-months of effort. The result is a simple tool, augmenting the designer's capabilities, allowing designs to be created very quickly, and leaving difficult problems to the designer. This simplification is not without a price. This section discusses the main problems with Riot and the steps which are being taken to remedy those problems in Riot and in future systems.

4.1 Connection by Positioning

A fundamental problem with Riot is its treatment of connection as geometrical position, which leaves the burden of maintaining logical connection to the designer. Riot determines positions of instances and routes by the positions of connectors on instances, but once the instances are positioned to make the connection, the fact that the two pieces are connected is lost, and the user is free to move the pieces in whatever manner is desired. When the logical connection is lost, some of the designer's intent is lost, also.

Some logical attributes, such as the layers and orientations of the connectors, are checked when a connection is made. However, the existence of a connection is not remembered, so the connections can easily be inadvertently destroyed by further editing. In practice, connections are not often accidentally destroyed, and when they are destroyed, the error is fairly easy to detect and correct. However, the mere possibility of missed connections requires extensive checking by users and has severely limited the usefulness of Riot.

4.2 Modification of Leaf Cells

A major problem which was anticipated and experienced with Riot was that connections, once made, were not preserved. With an interactive system, there were no major problems with accidentally and unwittingly destroying connections, but this problem showed itself very strongly when cells were changed.

When an existing leaf cell is modified, the locations of connectors are often changed also, and since Riot's connections are positional happenstance and Riot allows instances to overlap, connections will no longer be made properly and no warning message will be generated. Thus, the user is forced to re-edit major portions of the chip by hand to re-connect parts of the chip.

Riot includes an inexpensive solution to this problem, the REPLAY. Riot saves the commands given by the user and can re-run and editing session if some of the input files have changed. The replay file uses instance names and connector names to identify connections, and the positions are recalculated, thereby avoiding the problems with differently-shaped cells. The replay also enables users to recover an abnormally-terminated editing session or an accidentally-deleted file. The replay does not solve the problem of logical connection being destroyed during editing, but it does mitigate it. However, the replay is not an acceptable long-term solution to this important problem.

4.3 Conclusions

Riot has been used in the preparation of several small project chips, and appears to be a very good tool for such small designs. Riot reads many different data formats, simplifying the task of combining parts of designs from different systems. Riot provides a general interface to symbolic layout, allowing symbolic cells to be easily modified to fit into different environments on the chip, and on other chips. This greatly improves the usefulness of our symbolic layout systems.

The advantages and disadvantages of Riot have spurred further tool development in this area. Without further investigation, we can say that a tool of this type must maintain logical connections, a feature which is lacking in Riot. New graphical chip assembly tools are under construction at Caltech which will remedy this problem.

Even without maintaining connection, Riot has shown itself to be a powerful design aid. Riot allows designers to assemble test projects quickly with a minimum of interference from the tool. The simple connection commands are adequate to design large chips when the designer can use them as he wishes in different situations. Since the designer has the option at every point to choose the kind of connection operation, the design can be optimized at the level of assembly, not just at the level of layout. This kind of tool is a valuable addition to the integrated system designer's arsenal.

5 REFERENCES

- Calma (1980). "GDS II Product Specification", Calma Interactive Graphics Systems, Sunnyvale, CA.
- Johannsen, D. (1979). "Bristle Blocks: A Silicon Compiler", Proceedings of the 16th Design Automation Conference, 1979.
- Lang, D. (1979). "LAP: A SIMULA Package for IC Layout", Computer Science Department Technical Report, California Institute of Technology, 1979.
- Mosteller, R. (1981) "REST -- A Leaf Cell Design System", Master's Thesis, California Institute of Technology, 1981.
- Rowson, J.A. (1980). "Understanding Hierarchical Design", PhD Thesis, California Institute of Technology, 1980.
- Sproull, R. and Lyon, R. (1980). "The Caltech Intermediate Form for LSI Layout Description" in C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, New York, 1980.
- Trimberger, S. (1980). "The Proposed Sticks Standard", Computer Science Department Technical Report, California Institute of Technology, 1980.